Draft

Managing Distributed Systems with Smart Subscriptions

Robert E. Filman

Research Institute for Advanced Computer Science
NASA Ames Research Center
Mail Stop 269-1
Moffett Field, California 94305
rfilman@arc.nasa.gov

Diana D. Lee

Caelum Research Corporation
NASA Ames Research Center
Mail Stop 269-1
Moffett Field, California 94305
ddlee@arc.nasa.gov

Abstract

We describe an event-based, publish-and-subscribe mechanism based on using "smart subscriptions" to recognize unstructured events. We present a hierarchy of subscription languages (propositional, predicate, temporal and agent) and algorithms to expedite the recognition of subscriptions interested in particular events. This mechanism has been applied to the management of distributed applications.

Introduction

This work arose in the context of developing a framework (the Object Infrastructure Framework, or OIF) to simplify creating distributed applications. That project developed technology to endow distributed systems with better reliability, security, quality of service and manageability. OIF extended the standard mechanisms of distributed object technology (e.g., CORBA, Java RMI) to include discrete *injectors* on the communication paths between components. Injectors effectively wrap components and can manipulate and extend their communications. Because injectors are discrete, uniform objects, OIF is able to provide utilities to organize and manage them. One of the elements of the OIF is Pragma, a language for mapping between the specification of desired properties of a system and the consistent insertion of injectors that implement those properties on the methods of objects. OIF also includes a way for injectors to communicate, allowing injectors to annotate ordinary calls with additional information. By injecting the behavior for error recovery, redundancy, security checks, intrusion recognition, priority queue management, and so forth, OIF makes substantial progress in separating and simplifying

the first three classes of requirements: reliability, security and quality of service [Filman00].

However, the fourth, manageability, shows a certain resistance to a pure injection approach. What is manageability? Following the standard of the computer networking field [Stallings93], we identify five elements of manageability:

- **Configuration management**: Making sure the appropriate elements are in the appropriate places.
- **Fault diagnosis**: Monitoring and debugging a system by detecting conditions (complex and compound events) that indicate potential problems.
- **Intrusion detection**: Recognizing patterns that indicate a potential intrusion attempt (for example, a succession of unsuccessful logon attempts by a particular user with passwords selected from a dictionary).
- **Performance analysis**: Dynamically reporting information regarding the efficiency of various activities.
- **Accounting**: Tracking and logging patterns of interesting usage and events.

Of these five, the first proves amenable to a pure injector approach—configuration injectors can annotate requests with the local configuration, to be vetted by the correspondent. On the other hand, the others present something of a hybrid problem. Fault diagnosis can be aided by injectors reporting interesting calls and returned values; intrusion detection, by reporting suspicious events; performance analysis, by reporting the time requests were received and the time taken to process them; and accounting by reporting things worth billing. Injectors can report on management issues, but some additional structure is needed to implement manageability. That is, injectors provide a locus for recognizing and reporting interesting events, but not a structure for what to do with these events. This suggests two issues: (1) declaring to whom should an interesting event be reported and (2) defining what makes an event interesting.

One architectural approach would be (1) to have every recipient interested in some set of events inform every possible generator of such events of its interest, and (2) to make every generator of events responsible for informing every interested recipient on event occurrence. Such an architecture has several severe limitations, the most critical of which is that it requires too much knowledge of the structure of a system in too many

places. For example, changes in the organization of event producers need to be reflected in every event consumer. Direct connection between producers and consumers also has the potential for slowing down the critical activities of the producers to perform the event management, which might be as trivial as bookkeeping.

A solution to these problems is to position an intermediary between producers and consumers, an *event channel*. Producers funnel their events to the event channel. Consumers describe to the event channel which events interest them. The event channel is responsible for forwarding the appropriate events to the interested consumers. This notion of event channel also goes under the rubric "Publish and Subscribe." Event consumers "subscribe" to the (interesting) events "published" by event producers.

This work is devoted to the question, "How can we code event channels so that exactly the right events are delivered to the appropriate consumers?" Here we take a bit more inspiration from the network management experience. There are two standards for network management protocols: the Simple Network Management Protocol (SNMP) and the Common Management Information Protocol (CMIP) [Stallings93]. Roughly, SNMP is a simple low-level protocol that provides an awkward interface for probing the state of a network device. CMIP is a richer, higher-level protocol that allows more expressive data structures and the ability to have devices independently generate events on interesting occasions. CMIP is obviously the superior environment, with the minor caveat that, experientially, CMIP's reporting consumes 90% of the network bandwidth [Filman97].

We seek rich expressiveness without overwhelming communication cost. This can be accomplished by directing events only to interested parties. Although inspired by network management, we're doing this at the software component level, not the hardware device level (though conceptually, hardware devices can be seen as kinds of software components.) We're working under the assumption that communication costs far more than processing. Hence, it is better to expend effort checking that the communication is desired than to communicate volumes of uninteresting data. Of course, local processing isn't quite free, either. The are thus addressing two issues: (1) what *subscription languages* allow consumers to precisely express interesting events, and (2) which

algorithms allow event channels to organize the subscription space so as to efficiently recognize subscribers interested in particular events.

Architecture

Languages like CORBA and Java have popularized the notion of *interface.* An interface describes a set of behaviors (methods). An actual implementation class can support an interface by implementing the methods of the interface. The interface acts as a pseudo-type: variables can be declared to be of the interface's type and objects whose class supports the interface are recognized as being in it.

The two relevant interfaces for this discussion are event *consumers* and *event channels*. A consumer is an object to which one can *publish* an event (encoded as a string). A consumer is entitled to do whatever it wants with an event. The simplest consumer might just print the event on a debugging screen, write the event to a log file, or update a database with some salient facet of the event.

Any object can be an event *producer* by composing a string that is a good event representation and invoking publish on some consumer. (Producers are not, of course, structurally defined elements, as object-oriented systems do not regard the callers of particular methods as noteworthy groups.)

The interface *event channel* extends consumer with a *subscribe* method. Subscribe takes

(1) a reference to a consumer,

(2) a description, in some *subscription language,* of the set of events interesting to that consumer,

(3) a description of what about the existing event and environment is to be reported to the consumer (that is, the structure of an event to publish to the consumer), and

(4) optional signature information (discussed below) that can be used to optimize subscription algorithms.

Subscribe returns a ticket for managing the subscription. Using that ticket, the subscriber can modify or cancel a subscription. Event channels also include a method for obtaining

the closure of the set of subscriber interests—that is, a subscription that describes the union of all the channel's subscriptions. These relationships are illustrated in Figure 1.

Event channels, being consumers, also have a publish method. The implementation of publish in an event channel considers the new event in light of the existing subscriptions (and, perhaps, past events) and publishes that event (or some derivative of the event) to every consumer whose subscription matches the event.

In OIF, event producers can use not only the local arguments of their calls but also information from the thread's *environment* in deciding if a particular event is worthy of publication. OIF arranges to have the salient elements of this environment copied as part of the annotations of ordinary calls [Filman00]. For example, a process could tag a particular call with some special symbol and recognize processes created as consequences of that call as retaining that symbol in their environment.

In OIF, every virtual address space has one globally (within that address space) well-known event channel. Any application or injector that has an event to report can publish to that event channel. Any consumer that wishes to receive events can subscribe to its local event channel. Event channels on different virtual machines can subscribe to each other. In this way, the publish and subscribe mechanism becomes distributed, while the most appropriate local decisions are made about whether to distribute an event (Figure 2).

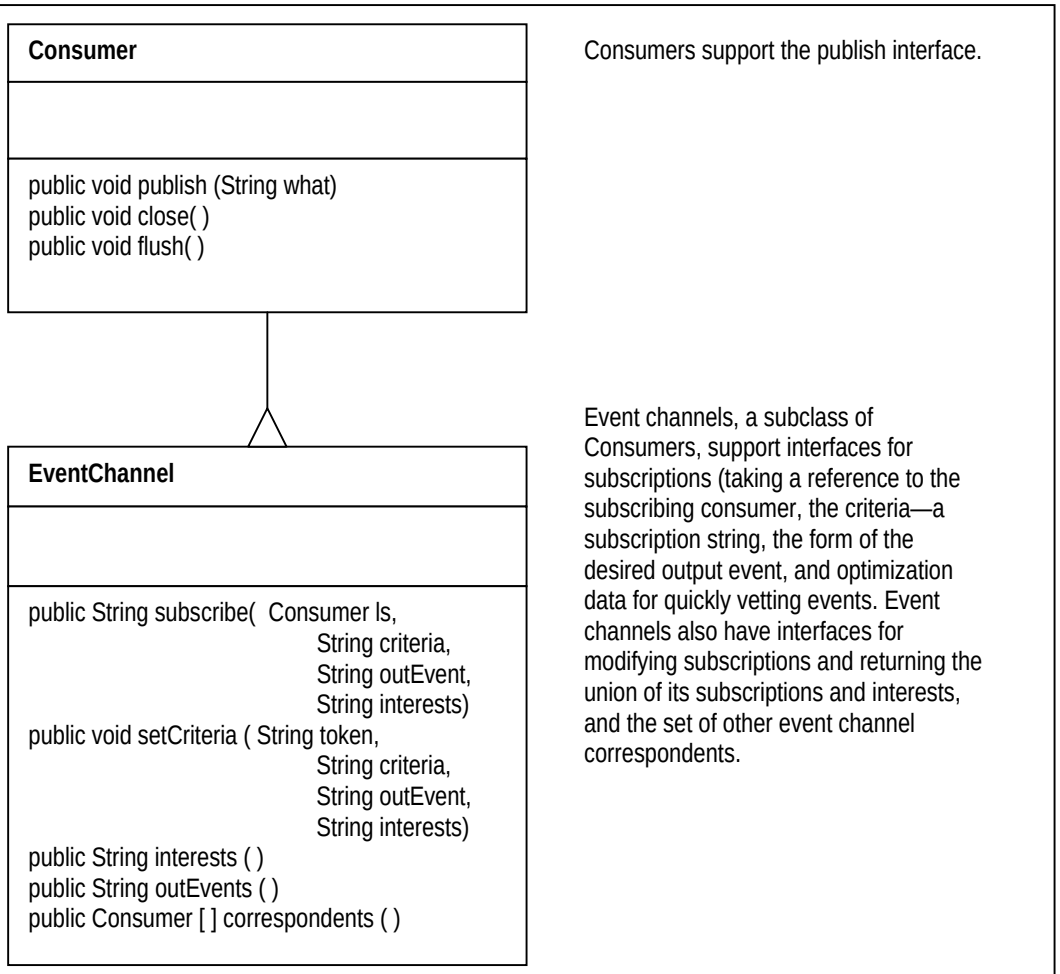

Figure 1. Consumer and EventChannel interfaces

How do the various virtual machines become aware of each other? OIF offered two different mechanisms. The first was to push this issue to the system architect. References to event channels could, after all, be listed on the namespace system, advertised on traders, and communicated in the body of messages. In this style, the application was responsible for setting up the event channel network and arranging appropriate subscriptions among the nodes of that network. Alternatively, the underlying system could arrange, as part of the annotations of communications, to distribute the knowledge of the existence of the caller's event channel. This has an interesting "event horizon" event—knowledge of channels interested in the effects of an action would travel as fast as that action and its consequences. In either case, the architect needed to ensure the "tree-like" distribution of every kind of event.

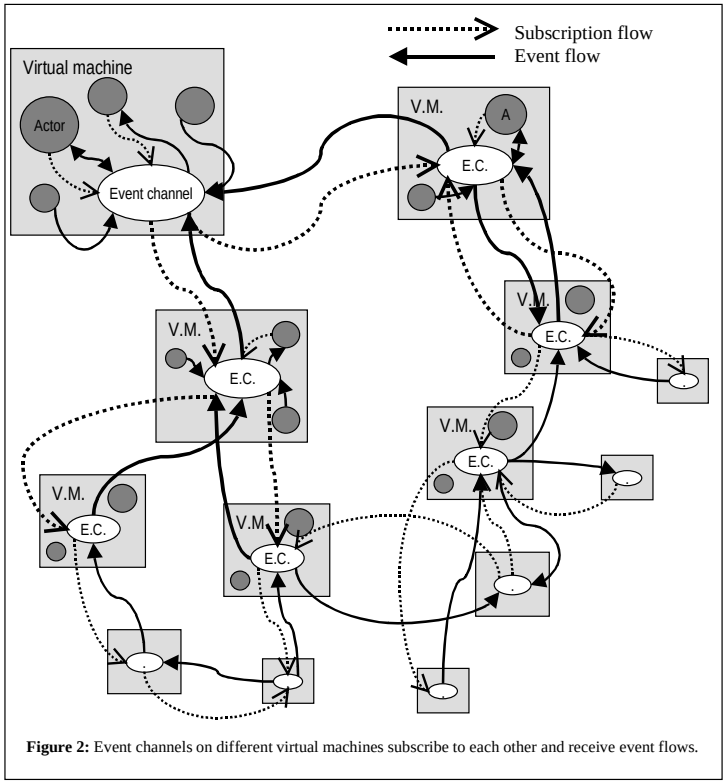

Figure 2: Event channels on different virtual machines subscribe to each other and receive event flows.

Events

Event systems usually support one of two different kind of events. Most event systems, in the structured tradition of strongly-typed programming languages, define classes of events as records, where the class structure is universally known. In such a model, the manipulator of an event knows exactly the fields of the record. Often the subscription language consists of merely describing interest in all records of a given class or subclass. This has the advantage of giving the programmer a reliable set of information on which to build—if I have an event of type *e*, I know it has fields *x*, *y*, and *z*. It has disadvantage of requiring too much commonly shared information, both in space and time. We do not want to demand that every event channel have knowledge of all kinds of events or even to posit the existence of an event definition repository. We expect the event structure to change, both as temporary event types are created to answer the questions of debugging and as new event types are created as part of the system evolution.

OIF takes the opposite tack, in the tradition of dynamic languages such as Lisp. In OIF, events are property sets (name–value pairs), without system restrictions (or promises) as to the existence of any particular name–value pair in any particular event. OIF provides a marshaling mechanism for converting event structures to strings for transmission, and an unmarshaling mechanism for reinflating them back to the property–value pairs. Thus, the string event representation:

"userid: Fred; time: 12:40:18; type: error; message: read unhappy maknam"

would translate into an event object with four properties (userid, time, type and message) with the corresponding (string) values. (The interpreter is responsible for doing data conversion for numeric operators. There are specific notations for strings that represent remote object references and values that are themselves events.)

For debugging and system evolution, the property approach allows us to introduce new event fields into a running system. In terms of subscription languages, reference to the fields of events is straightforwardly uniform. This has the further virtue that no common understanding of event structure definitions is required across the distributed system. It has the corresponding disadvantage that we lack compile-time checks that structures will have properties not explicitly demanded in subscriptions.

Subscription languages

A goal of this work was to minimize uninteresting communications. Broadly, this suggests a richly expressive subscription language, where a subscriber can precisely describe which events are of interest. However, the richer the subscription language the more effort is involved both at coding time in creating the subscription interpreter and at run time in deciding if a particular subscription is satisfied by a given set of events. In OIF, we created a series of subscription languages of increasing expressiveness. In this we are reminded of the hierarchies of automata, formal language grammars, and logics, where successively elements extend the expressibility of simpler mechanisms, often at the cost of greater complex computability. (In practice, in both formal language theory and OIF, these structures are not always strictly hierarchical.) In OIF, we have four subscription languages: *propositional*, *predicate*, *temporal* and *agent*.

The **propositional language** deals solely in the existence of properties of events. A subscriber can express interest in A, B, and C, and any event that mentioned (as properties at the top-level) A, B and/or C matches. In some sense, the propositional language is a property-list generalization of the class hierarchy of conventional subscription languages. An event with properties A, B, and C can be viewed as being in the classes "HasA," "HasB," and "HasC." A subscription would be the disjunction of such classes.

The **predicate language** provides a way to refer to the values fields of events (and subfields of contained events), constants, and values from the environment; and to combine these values with relations (e.g., "less than") and propositional connectives (e.g., "or," and "not") to form a logical well-formed formula. Using a Cambridge-prefix syntax, a subscription looking for error or warning events for user Joe would be expressed as

```
(and  (or  (= type 'error)
           (= type 'warning))
      (= user 'Joe))
```

where the quotes designate literals.

The **temporal language** loosens the prior restriction to single events. The propositional and predicate languages reference a single event at a time and, as a default, forward that event to the consumer. The temporal language allows for expression of relations among several events. Thus, one can talk about the existence of events E1, E2, and E3, such that E1 has occurred before E2, which occurred before E3, and which share a common user. We adopted the language of JESS [Friedman-Hill98], a RETE-based forward chaining, rule-based expert-system shell for our temporal language. (The "?" variables here are to be understood as being unified, much as in Prolog.) Using JESS, the preceding subscription would be expressed as:

```
(event (time ?t1) (userid ?u1))                    (1)
(event (time ?t2) (userid ?u2))
(event (time ?t3) (userid ?u3))
(test (< ?t1 ?t2 ?t3))
(test (eq ?u1 ?u2 ?u3))
```

In this example, the `(event (time ?t1) (user ?u1))` statement requires that an event (`E1`) include the time and userid properties where `?t1` and `?u1` represent the

values of the time and userid respectively. The statement `(test (< ?t1 ?t2 ?t3))` requires `?t1 < ?t2 < ?t3`; the statement `(test (eq ?u1 ?u2 ?u3))` requires the userids of `E1`, `E2`, and `E3` be equal.

In practice, memory is not infinite. Any practical temporal event matching mechanism needs a policy about discarding "no longer relevant" events—for example, discarding events that haven't matched a rule element within the last *X* minutes. Similarly, the system needs to define a concurrency semantics relating the timing of subscriptions and events. In OIF, we adopted the naïve approach that only the most recent *n* events were promised to be available for matching and the new subscriptions might recognize old events.

The **agent language** carries the implication of the Cambridge-prefix form to its logical extension. That is, subscriptions are themselves programs, invoked by event occurrences and able to examine the local event repository. This then becomes a mechanism for distributing agents throughout a system. Since we have not yet implemented an agent language, we have little to say about them except to note their existence at the top of the language hierarchy and their straightforward implementation with any of the standard Lisp interpreters.

In operational terms, the subscription of a subscribe method expected a string. The event channel would parse this string with respect to the particular language. In our implementations, we used Cambridge prefix form as the grammatical substrate of the various subscription languages, as it is the simplest-to-parse recursive language.

Event channel algorithms

We turn to the issue of efficiently implementing the subscription mechanism in event channels. Naïvely, the event channel could sequentially go through its subscriptions, checking each for satisfaction. We note five possible algorithmic improvements on this behavior: *sig, memo, lattice, compile* and *Rete*. (We have implemented all but the fourth.) Which is best? The optimal subscription channel algorithm is a function of the expected distribution of events and subscriptions. Some algorithms take advantage of an expected variety in the published events, while others do better on related or repeated event types.

Similarly, the amount of effort expended when a new subscription is received can be worthwhile only given a particular frequency of subscription changes.

Sig, memo and lattice rely on recognizing the *signature* of subscriptions. The signature of a subscription is set of event properties demanded by the subscription. The signature of

```
(and  (or  (= type ‘error)
           (= type ‘warning))
      (= user ‘Joe))
```

is

```
{type, user}
```

while the signature of

```
(and  (or  (= type ‘error)
           (= status ‘warning))
      (< 3 delay)
      (= session ‘g0043))
```

is

```
{delay, session}
```

Sig

On receiving an event, Sig determines the properties mentioned in that event. Let us call that set *fields*. Before evaluating the subscription of each subscriber, it checks to see if the *signature* of that subscription is a subset of *fields*. As both the signature and the fields are represented with bit vectors, the subset comparison can be quick (as long as the application is not wantonly generating new properties). That is, Sig is a fast way of establishing that a given event is not of interest to a subscription because it is missing some necessary properties. Sig is appropriate for applications that generate a variety of different events and use computationally complex subscriptions.

Memo

Memo, like Sig, computes the fields of an event and compares them to the signatures of subscriptions. It extends this by constructing a memoization table mapping fields to successful signatures. Having discovered, for example, that fields {delay, session, type} satisfy the signatures of subscriptions *X*, *Y*, and *Z*, Memo stores the mapping {delay,

session, type} → {*X*, *Y*, *Z*} in its memo table. When another event with that same fields occurs, Memo finds the stored value and checks only the subscriptions of *X*, *Y* and *Z*. On subscription updates, Memo needs examine the power set of the signature of the changed subscription (or at least those items not in the intersection of the power sets of the old and new signatures), updating the corresponding memo values. In practice, it may be easier to clear the entire memo table than to perform this computation (which might have a healthful, purgative effect, anyway.)

Memo is useful for situations where the subscription set changes slowly and events with the same fields occur repeatedly.

Lattice

Lattice extends Memo with a notion of subsumption. That is, if subscription *X* has signature {A, B}, and subscription *Y* has signature {A, B, C}, then if event E is not of interest to *X*, it is certainly not of interest to *Y*.

In general, the signatures of subscriptions form a lattice with respect to subset (Figure 4). The lattice algorithm constructs the (sparse) lattice as a data structure. We'd like to be able to move up the lattice, discarding the entire superstructure over a subsuming node if the "of interest" test fails. Unfortunately, we haven't been able to determine an efficient way of doing this—sorting through the multiple connections of the lattice seems to be more work than sequentially checking signatures. The current lattice algorithm works by "flattening" the lattice to a single path. That is, of all the possible subsumptions in the lattice, we select a subset such that each node (except the root) is subsumed by exactly one other. In implementation terms, this is expressed as a skip table. All subscriptions with the same signature are grouped as a set. The set are arranged in a sequential table, number 0, number 1, and so forth. Associated with this table is another "skip table." Failure to meet the signature test on item *i* in the table allows one to consult the skip value, call it *j*, and to continue searching signatures starting at *j*. All the items between *i* and *j* have been subsumed by the failure at *i*. Conceivably, a dynamic version of lattice could reorder the flattening sequence in response to the historical pattern of events.

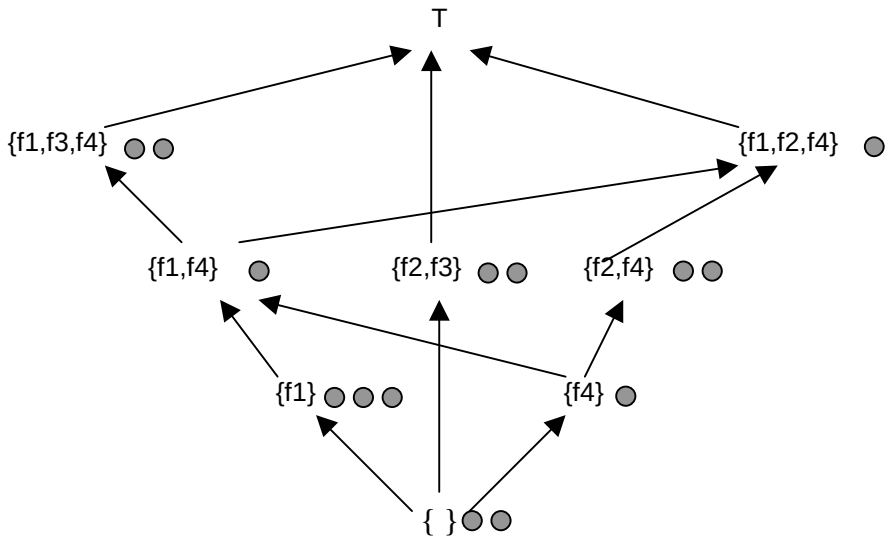

Figure 4. A partial lattice for several signatures. The small circles represent subscriptions associated with that signatre.Starting at bottom, if an event fails a containment test (say at {f1}) then it will certainly fail all nodes reachable below {f1}, that is, {f1,f4}, {f1, f2, f4}, {f1, f3, f4} and T.

Lattice handles subscription change more easily than Memo, and is most appropriate is when there is a lot of subsumption in the subscription structures.

Compile

Each subscription can be viewed programmatically: if the subscription condition is met, then perform the forwarding action. Compile treats the entire subscription set as a program by (1) sequencing the subscriptions, and (2) performing arbitrary compiler optimizations on the resulting program. In particular, elements such as common sub-expressions can be moved forward so as to be computed only once, tests such as (> x 3) can be placed so as to shadow (> x 7), and subsumptions can be realized by moving subsumed rules into the then-parts of more general subscriptions.

Compile is most appropriate for a relatively static subscription set that contains a large number of common sub-expressions.

Rete

The first four algorithms deal with single-event subscriptions—that is, where the subscription refers to just one event. What of subscriptions in temporal languages, that can say things like “when five password hacking events have occurred on an account recently, warn …”? For such subscriptions we turned to Rete.

Rete is a standard AI algorithm that addresses the problem taking a set of patterns and matching them against a set of events. (In most AI applications, learning a new fact is an event.) For most algorithms, the combinatorial nature of comparing large sets of patterns to large sets of objects becomes a challenge in efficiency. However, Rete is particularly successful in being time-efficient. Rete works by storing the partial matches of conditional statements in a network structure. When additional events come in, they are matched against the appropriate partial matches and more complete partial matches are moved further through the network.

We use Rete by mapping OIF-subscription conditions to Rete patterns and OIF-events to Rete events,. We implemented an OIF interface to the Java Expert System Shell (JESS) implementation of Rete. Using the JESS language, the general form of a subscription becomes a list of conditions followed by a list of actions to take when the conditions are met by a set of events. The conditions can express ideas such as the “existence of events E_1, E_2, and E_3, such that E_1 has occurred before E_2 that occurred before E_3, that share a common user” (as we illustrated in example (1)). Or conditions can be expressed as an algorithm encapsulated in a computer program or function. For example, if there is a program or function whose signature is

```
DetectHackingEvents(int number, int timeInterval),
```

one can make Rete aware of its existence by using an OIF/JESS interface. Then one can express the condition “when five password hacking events have occurred on an account in the last five minutes” as

```
DetectHackingEvents(5,5)
```

Similarly, the list of actions that one can ask Rete to perform when a set of conditions is met can be as simple as a printing a message or can be as complicated as an algorithm encapsulated in a computer program or function.

The major weakness of Rete is its appetite for storage space—the structures used to store the network of partial matches can become large. OIF includes an interface to allow cleanup of old items in these structures.

Applications

We have implemented the event channel mechanism described here in the OIF distributed computing framework, and applied it in a demonstration application [Lee98]. That application implements a simulation of a distributed, competitive network management application. It uses injectors to achieve quality of service (i.e., real-time performance), manageability and security. It used the event mechanism to dynamically drive "inspector" user interfaces. The event mechanism also proved critical in debugging the application, particularly as the injector mechanism could be set to generate events on every remote invocation. Events could then be selectively scanned to get a trace of interprocess calls, and this trace could be transparently directed to both visible graphic user interfaces and textual logs.

In general, in OIF one can arbitrarily and dynamically modify the injectors of proxies or set the default behavior of a set of proxies to include a particular injector. By making an injector that generates trace events and applying that injector appropriately, the event mechanism can be made to track the patterns of interprocess calls in the system.

Related work

Event models

Event channels are a realization of Arbab's *Ideal Worker Ideal Manager* (IWIM) model of anonymous communications [Arbab96]. Workers are free of having to know who consumes their production; managers are free of having to know the specifics of who produced their consumables.

In the taxonomy of Coordinated Computing [Filman84], the OIF event-channel mechanism is a problem-solving, process approach that supports dynamic processes; unbuffered, asynchronous communication with unidirectional information flow; passive communication control; pattern-matched reception and (effectively) broadcast sending,

with no explicit notions of time, fairness or failure, and a strongly pattern-directed invocation mechanism.

From the point of view of the Framework for Event-Based Software [Barrett96], OIF's event mechanism uses point-to-point, application-to-application communication. Modules have no explicit specification of their interfaces. It supports dynamic system modification and allows fully abstract naming. Our publishers are Barrett's informers; our consumers, listeners; and our event channels, routers. The subscription mechanism effectively serves to do message transformation. We posit no delivery constraints beyond the underlying distributed object framework. The local event channel on each virtual machine serves as a group.

Rosenblum and Wolf [Rosenblum97] describe a seven-model framework for event observation and notification. Within that framework, our publishers are the invoker objects and subscribers, the objects of interest. Events are the explicitly generated by invoking the send event action, naming is implicit in the naming of event fields (the property-based model), observation is by explicit subscription, information is by the action of a subscription, pattern abstraction and filtering is by the pattern part of the subscription language, and the partitioning arises naturally from the set of subscriptions made. We have no explicit time model, notification is by distributed object technology calls, and the resources for sorting through subscriptions are provided by the sender and the intermediary event channels.

Event implementations

Bates [Bates95] argues for using a rule-based publish and subscribe system to debug heterogeneous, distributed systems. Primitive events are defined and source code is annotated so that the executing program generates event instances. When a user-defined model of behavior is matched to the event stream, a high-level event is recognized. Behaviors are modeled in terms of event classes and their relationships, and may be composed to form higher-level behavior models. Bates also uses a rule-based engine for complex event detection, fairly similar to Rete, though independently discovered.

The Elvin project is a publish-subscribe service that delivers notifications on the basis on the event's content [Segall97]. It has an event subscription language that allows

subscribers to place some constraints over the notifications, flexible definition of events that allow developers to define events as required, dynamic definition of event types, and allows for the creation of new events based on old events. Elvin also introduces the idea of *quenching* that "allows event producers to receive information about what consumers are expecting of them so that they need only generate events that are in demand." This corresponds to our notion of event channels being able to provide the union of their subscriptions. In contrast to Elvin, which has a single centralized event channel, OIF's event channels are distributed. Lacking a centralized, potential bottleneck, OIF is thus scalable, though the architecture leaves open to the user arranging the actual connectivity.

The Ariadne Debugger in TAU stores an execution history graph of events and allows the subscriber to specify patterns using a simple subscription language that is capable expressing temporal relations among several events but unable to express other simple prepositional or complex relations among events [Shende96]. To "compensate" for where the language lacks, Ariadne "provides a scalable, spread-sheet like interface for exploring match trees."

The CEDMOS project architecture is composed of event-producers and event-consumers that are connected through event-transformers. "The event transformers convert streams of incoming events into different streams of events, which are …of interest to the event-consumers" [Baker98]. To facilitate the event transformers, a graphical tool facilitates the definition of complex event from simple events.

Brant and Kristensen apply events to web-based notification. Their architecture includes the notions of annotated lists, a well-worked-out datatype mechanism and a good notion of the idea of filtering [Brant97]. Intermetrics [Ress98] describes a design for applying events to doing debugging of distributed, component-based products. Luckham and Frasca apply event patterns, causal histories, filtering and aggregation to provide higher levels of abstractions for managing distributed systems [Luckham98]. The notion of spreading communication about information providers is seen in the lookup service provides in Jini [Waldo99]. Jini lookup service providers inform each other of the lookup service providers they know about.

Summary and discussion

We have discussed the publish and subscribe mechanism in the Object Infrastructure Framework. This mechanism has proved to be a powerful tool in debugging and managing distributed systems, supporting functions such as fault diagnosis, intrusion detection, performance analysis, and accounting. Key elements of this work are existence within a framework that provides a continuing environmental context, the use of unstructured events, rich subscription languages, and selectable and efficient algorithms for subscription resolution. Topics for further work include (1) subscription-forwarding mechanisms that do not require tree-like branching, (2) security mechanisms for subscriptions and event channeling (including the ability of an event generator to limit who could notice his events), (3) quantifying the actual performance of different event-channel algorithms in realistic cases, (4) implementing agent subscription languages, and (5) implementing subscription compilation.

References

1. Arbab. F. The IWIM model for coordination of concurrent activities. *Coordination '96, Lecture Notes on Computer Science, vol.1061*, P. Ciancarini, and C. Hankin (Eds.), Springer-Verlag, New York, 1996.

2. Baker, D., Cassandra, A., Rashid, M. CEDMOS: Complex Event Detection and Monitoring System. Microelectronics and Computer Technology Corporation, 1998.

3. Barrett, D. J., Clarke, L. A., Tarr, P. L., and Wise, A. L. An Event-Based Software Integration Framework. *ACM Transactions on Software Engineering and Methodology 5*, 4 (October 1996) 378–421.

4. Bates, P. C. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems 13*, 1 (February 1995), 1–31.

5. Brandt, S. and Kristensen, A. Web Push as an Internet Notification Service, W3C Workshop on Push Technology, (Boston, Massachusetts, September 1997), http://keryxsoft.hpl.hp.com/doc/ins.html.

6. Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. Inserting Ilities by Controlling Communications. To appear in *Communications of the ACM*, 2000.

7. Filman, R. "The Arachnoid Tourist: Managing a Spider's Net," *IEEE Internet Computing, Vol. 1*, No. 5, October, 1997, pp. 50–51.

8. Filman, R. E., and Friedman, D. P. *Coordinated Computing: Tools and Techniques for Distributed Software*. McGraw-Hill, New York, 1984.

9. Friedman-Hill, E. J. Jess, The Java Expert System Shell. DANS98-8206 Distributed Computing Systems Sandia National Laboratories. Livermore, CA, (September 1998), http://herzberg.ca.sandia.gov/jess

10. Forgy, C. L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence 19*, 1 (1982) 17–37.

11. Lee, D. and Filman, R. Verification of Compositional Software Architectures. Workshop on Compositional Software Architectures, Monterey, California, January 1998. http://www.objs.com/workshops/ws9801/papers/paper096.doc.

12. Luckham, D. C. and Frasca, B. Complex Event Processing in Distributed Systems. Stanford University Technical Report CSL-TR-98-754 (March 1998), ftp://pavg.stanford.edu/pub/cep/fabline.ps.Z

13. Ress, J. Intermetrics' Owatch Debugging Technology for Distributed, Component-Based Systems. OMG-DARPA-MCC Workshop on Compositional Software Architectures (Monterey, California, January 1997) http://www.objs.com/workshops/ws9801/papers/paper058.html.

14. Rosenblum, D. S., and Wolf, A. L. A Design Framework for Internet-Scale Event Observation and Notification. *Proceedings of the Sixth European Software Engineering Conference / ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering* (September 1997), 344–360.

15. Segall, B. and Arnold, D. Elvin has left the building: A publish/subscribe notification service with quenching. *Proceedings of AUUG97* (Brisbane, Queensland, Australia, September 1997).

16. Shende, S., Cuny, J., Hansen, L., Kundu, J., McLaughry, S., and Wolf, O. Event and State-Based Debugging in TAU: A Prototype. *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools* (Philadelphia, May 1996), 21–30.

17. Stallings, W. *SNMP, SNMP-2, and CMIP: The Practical Guide to Network-Management Standards*. Reading MA: Addison-Wesley 1993

18. Waldo, J. The Jini architecture for network-centric computing, *Comm. ACM 42*, 7 (July 1999), 76–82.